
Toward High-Performance Computational Chemistry: II. A Scalable Self-Consistent Field Program

**ROBERT J. HARRISON, MARTYN F. GUEST, RICK A. KENDALL,
DAVID E. BERNHOLDT, ADRIAN T. WONG, MARK STAVE,
JAMES L. ANCHELL, ANTHONY C. HESS, RIK J. LITTLEFIELD,
GEORGE L. FANN, JAROSLAW NIEPLOCHA, GREG S. THOMAS,
and DAVID ELWOOD**

Pacific Northwest Laboratory, P.O. Box 999, Richland, Washington 99352

**JEFFREY L. TILSON,* RON L. SHEPARD, ALBERT F. WAGNER,
IAN T. FOSTER, EWING LUSK, and RICK STEVENS**

Argonne National Laboratory, Argonne, Illinois 60439

Received 21 June 1994; accepted 10 April 1995

ABSTRACT

We discuss issues in developing scalable parallel algorithms and focus on the distribution, as opposed to the replication, of key data structures. Replication of large data structures limits the maximum calculation size by imposing a low ratio of processors to memory. Only applications which distribute both data and computation across processors are truly scalable. The use of shared data structures that may be independently accessed by each process even in a distributed memory environment greatly simplifies development and provides a significant performance enhancement. We describe tools we have developed to support this programming paradigm. These tools are used to develop a highly efficient and scalable algorithm to perform self-consistent field calculations on molecular systems. A simple and classical strip-mining algorithm suffices to achieve an efficient and scalable Fock matrix construction in which all matrices are fully distributed. By strip mining over atoms, we also exploit all available sparsity and pave the way to adopting more sophisticated methods for summation of the Coulomb and exchange interactions. © 1996 by John Wiley & Sons, Inc.

* Author to whom all correspondence should be addressed

Introduction

We discuss the design of scalable parallel algorithms that emphasize the cost-effective use of both processors and memory. Efficient self-consistent field (SCF) computations are required for our research and for more accurate theories, and the direct¹ closed-shell SCF method for molecules is used as an example. SCF is representative of more sophisticated theories in using irregular data access patterns and accumulating results into large data structures. However, with current approaches, the computational cost of SCF grows at such a high rate with respect to problem size that even teraFLOP/s computers[†] would permit only modest growth in problem size. To make more substantial improvements, it is necessary to incorporate more sophisticated algorithms with lower growth rates—for example, replacing an $O(N_{\text{atom}}^4)$ algorithm with one with an effective scaling of $O(N_{\text{atom}})-O(N_{\text{atom}}^2)$.²⁻⁵ Parallelization of such algorithms will be the subject of future publications.

To achieve this level of scalability will require the use of massively parallel processors (MPPs) and alternative numerical approaches. A recurring theme in our algorithm and software design and analysis is to develop techniques and tools that permit the development of scalable applications with explicit parallel constructs with only a minimal amount of extra effort.

The first parallel SCF programs⁶⁻¹² were based on the replicated data model, in which each processor in the parallel system held its own complete copy of the Fock and density matrices. While this approach is simple and performs acceptably well for moderate-sized chemical systems, it does not scale well to either large chemical systems or massively parallel computers. Current parallel computers have only enough memory per node to replicate matrices of size 100 to 1000 square, and the high cost of memory relative to processors precludes adding memory. Instead, it is both necessary and cost-effective to distribute the Fock and density matrices across the memories of all processors.

Previous distributed SCF algorithms^{13,14} had the advantage of being based on regular communication patterns that were compatible with conven-

tional two-sided message passing semantics (cooperative sender and receiver). They suffered, however, from inefficiencies arising from performing redundant computation, limited scalability, and/or large process waiting times caused by load imbalance and frequent synchronization. In addition, development of these parallel programs based on message passing required significant effort beyond that of developing a sequential program with similar functionality. We wish to understand not merely how to write a fast and scalable SCF program, but to isolate the principles that make this task easier and then apply those ideals to other algorithms of computational chemistry.

In this article we continue our previous work¹⁵ and explore an approach based on a novel communication strategy. The Fock matrix construction is parallelized using a dynamic load-balancing approach, as with the replicated data model.¹² However, the Fock and density matrices are distributed across processors and are accessed during the Fock build using a one-sided remote-fetch/store/accumulate communication paradigm that is conceptually similar to shared memory. (We use the term *asynchronous global arrays* to describe this capability, which is discussed in more detail later in this article.) This approach requires no extra computation (compared with the sequential algorithm) but is potentially communication intensive.

Parallel Replicated-Data Fock Matrix Construction Algorithm

The replication of the Fock and density matrices within each processor of a distributed memory parallel computer eliminates all communication during the two-electron Fock matrix contribution. Each processor computes part of the integral list, adding the parts into its own local matrix. Subsequently, the complete Fock is obtained by combining the partial matrices with a global summation operation. The algorithm is perfectly parallel (assuming some rudimentary load balancing^{8,12,16}), and the global summation of the Fock matrix can be done efficiently (e.g., on the Intel Touchstone Delta, a global summation of 10^6 numbers takes about 3 s).

The poor scaling inherent in this approach is readily apparent. To hold two symmetric matrices of dimension N on each of P processors requires approximately $8PN^2$ bytes of memory. The size of calculation is constrained by the memory on each

[†] Computers capable of 10^{12} floating point operations per second.

processor, and the cost of the machine is dominated by memory rather than processors. By abandoning replicated data algorithms, we remove the rigid constraint on problem size and enable the usage of machines with a far more cost-effective ratio of processors to memory, a point raised by Hillis¹⁷ in justifying the design of the CM-2.

Burkhardt et al.⁹ distributed the integral computation across many small Transputer-based nodes, keeping the Fock matrix on just one processor with a large memory. While this addresses the cost issue, the algorithm is intrinsically not scalable, since accumulation of the integrals into the Fock matrix is a sequential bottleneck. Fully distributed algorithms^{13,15} are required.

One key to understanding the design and performance of fully distributed algorithms is an understanding of how to manage efficiently the memory hierarchy of parallel computers. We consider such issues in the next section.

NUMA: Nonuniform Memory Access

NUMA refers to some regions of memory being more expensive to access than others. Consider, for instance, a standard RISC workstation. Its high performance stems from reliance on algorithms and compilers that optimize usage of the memory hierarchy formed by registers, on-chip cache, off-chip cache, main memory, and virtual memory. The programming of MIMD parallel computers (either shared or distributed memory) is united with the programming of sequential computers by the concept of NUMA. By focusing on NUMA instead of on the details of the programming model, parallel computation is seen to be different from sequential computation only in the essential difference of concurrent execution, rather than in nearly all aspects.

STRIP MINING OR BLOCKED ALGORITHMS

Consider the multiplication of two $N \times N$ matrices

$$C_{ij} = C_{ij} + \sum_{k=1}^N A_{ik} B_{kj} \quad (1)$$

A traditional vector supercomputer can fetch data from memory as fast as it can compute (which is why supercomputers are expensive—again, it is the memory for which you are paying). Thus, it is not too ridiculous to use the simplest loop

structure to implement the matrix multiplication (Fig. 1).

The total number of memory references in the inner loop is $2N^3$, which is the same as the operation count. With the slower memory of low-cost workstations, it is necessary to modify the algorithm so that most memory references are to blocks of the matrix kept in the fast cache. The blocking is achieved by partitioning or stripmining each loop (Fig. 2) into N_{block} blocks.

The traffic between the cache and processor is still $2N^3$, but the traffic between the cache and memory is now just $2N_{\text{block}}N^2$, a reduction by a factor of N/N_{block} . If N_{block} is adjusted so that this ratio significantly exceeds the ratio of the bandwidths processor to cache, and cache to memory, then the matrix multiply can proceed essentially unhindered by the slow memory (the cache must be big enough to hold the necessary block size). The issues are identical in parallel computation, except interprocessor bandwidth is compared with local memory bandwidth, and the latencies associated with memory reference are more significant and must be incorporated into performance models. The method we use to reduce communication in the parallel Fock matrix construction is identical to that used for this simple matrix multiplication example.

ONE-SIDED MEMORY ACCESS

No emerging standards for parallel programming languages (notably just High Performance Fortran, HPF-1; see ref. 18) provide extensive support for MIMD programming. The only truly

```

DO i = 1, N
  DO j = 1, N
    sum = Cij
    DO k = 1, N
      sum = sum + Aik * Bkj
    ENDDO k
    Cij = sum
  ENDDO j
ENDDO i

```

FIGURE 1. A simple matrix-multiplication algorithm.

```

DO iblock = 1, Nblock
  DO jblock = 1, Nblock
    Load block of C into cache

    DO kblock = 1, Nblock

      Load A and B blocks into cache

      DO i in iblock
        DO j in jblock
          sum = Cij

          DO k in kblock
            sum = sum + Aik * Bkj
          ENDDO k

          Cij = sum
        ENDDO j
      ENDDO i
    ENDDO kblock
  ENDDO jblock
ENDDO iblock

```

FIGURE 2. A strip-minded matrix-multiplication algorithm.

portable MIMD programming model is message passing, for which a standard interface has recently been proposed.¹⁹ It is, however, very hard to develop applications with fully distributed data structures using the message passing model.^{13,14} What is needed is support for one-sided access to data structures (here limited to one- and two-dimensional arrays) in the spirit of shared memory. With some effort, this can be done portably,²⁰ and in return of this investment we gain a much easier programming environment that speeds code development and improves extensibility and maintainability.

We also gain a significant performance enhancement from increased asynchrony of execution of processes.²¹ Message passing forces processes to cooperate (e.g., by responding to requests for a particular datum). Inevitably, this involves waiting for a collaborating process to reach the same point in the algorithm, which is only partially reduced by the use of complex buffering and asynchronous communication strategies. With a one-sided communication mechanism, each process can access

what it needs without explicit participation of another process, and all processes can operate independently. This eliminates unnecessary synchronization and naturally leads to interleaving of computation and communication.

Prototype Support for Distributed Globally Addressable Arrays

The requirements of the SCF algorithm (discussed later), the parallel COLUMBUS configuration interaction program,²² the second-order Møller-Plesset perturbation theory, and parallel coupled-cluster methods²³ led to the design and implementation of some preliminary tools²⁰ to support one-sided access to distributed one- and two-dimensional arrays. In this section we outline briefly the functionality provided by this library.

PROGRAMMING MODEL

The current globally addressable (GA) programming model can be characterized as follows:

- MIMD parallelism is provided using a multiprocess approach, in which all non-GA data, file descriptors, and so on are unique to each process.
- Processes can communicate with each other by creating and accessing GA distributed matrices and also (if desired) by conventional message passing.
- Matrices are physically distributed block-wise, either regularly or as the Cartesian product of irregular distributions on each axis.
- Each process can independently and asynchronously access any 2D patch of a GA distributed matrix (operations including *get*, *put*, floating point *accumulate*, and atomic integer *get-and-increment*) without requiring cooperation by the application code in any other process.
- Each process is assumed to have fast access to some portion of each distributed matrix and slower access to the remainder. These speed differences define the data as being local or remote, respectively. However, the numeric differences between local and remote access times is unspecified.
- Each process can determine which portion of each distributed matrix is stored locally. Ev-

ery element of a distributed matrix is guaranteed to be local to exactly one process.

Arrays may be dynamically created and destroyed. In addition, a set of linear algebra data-parallel operations have been developed on top of the primitive operations, including vector (e.g., dot product) and matrix (e.g., symmetrize, multiplication) operations. Interfaces are provided to third-party libraries (e.g., SCALAPACK[‡] using the GA primitives to perform necessary data rearrangement. The $O(N^2)$ cost of such rearrangement is observed to be negligible in comparison to that of $O(N^3)$ linear algebra operations. These libraries

random access to distributed arrays from within a MIMD parallel subroutine call tree, and reduction into overlapping regions of shared arrays.

The following code fragment uses the FORTRAN interface to create an n by m double precision array, blocked in at least 10 by 5 chunks, which is zeroed and then has a patch filled from a local array. Undefined values are assumed to be computed elsewhere. The routine `ga_create()` returns in the variable `g_a` a handle to the global array with which subsequent references to the array may be made.

```
integer g_a, n, m, ilo, ihi, jlo, jhi, ldim
double precision local(1:ldim,*)
c
call ga_create(MT_DBL, n, m, 'A', 10, 5, g_a)
call ga_zero(g_a)
call ga_put(g_a, ilo, ihi, jlo, jhi, local ldim)
```

The preceding code is very similar in functionality to the following HPF-like statements:

```
integer n, m, ilo, ihi, jlo, jhi, ldim
double precision a(n,m), local(1:ldim,*)
!hpf distribute a(block(10),block(5))
c
a = 0.0
a(ilo:ihi, jlo:jhi) = local(1:ihi-ilo+1, 1:jhi-jlo+1)
```

may internally use any form of parallelism appropriate to the computer system, such as cooperative message passing or shared memory.

SAMPLE CODE FRAGMENT

This interface has been designed in light of emerging standards. In particular, High Performance Fortran (HPF)¹⁸ will provide the basis for future standards definitions of distributed arrays in FORTRAN. The basic functionality described earlier (create, fetch, store, accumulate, gather, scatter, data-parallel operations) all may be expressed as single statements using FORTRAN-90 array notation and the data-distribution directives of HPF. What HPF does not currently provide is

The difference is that this single HPF assignment would be executed in a data-parallel fashion, whereas the global array put operation will execute in MIMD parallel mode such that each process may reference different array patches.

The Distributed SCF Algorithm

Our initial distributed SCF prototype code¹⁵ parallelized only the two-electron contribution to the Fock matrix construction. This effort was successful in that an acceptably efficient parallel Fock matrix construction was achieved. It was apparent, however, that the remainder of the program, which was still using replicated data methods, needed complete rewriting. There were several problems: The distributed data tools we had used did not accommodate efficient data-parallel operations; the distribution of tasks was overly complex, causing

[‡] SCALAPACK—scalable linear algebra package, code, and documents—available through `netlib`.

load-balancing problems; excess data were being moved in sparse problems; and the overall parallel scalability was not as good as desired.

The design of the current algorithm follows the analysis used for the foregoing matrix-multiply algorithm. The cost of accessing an element of the global density or Fock matrices must be offset by using this element multiple times. To achieve this reuse of data, we simply strip mine the fourfold loop over basis functions. Since the computation is a quartic function of the block size while the communication is only a quadratic function, small block sizes suffice to make the computation time dominate the communication time. This is the same principle independently used by Furlani and King,¹⁴ but in contrast to the complexities of their message passing algorithm, the support for one-sided access to distributed arrays enables a simple, efficient, and readily modified implementation.

We strip mine by grouping basis functions according to their (usually atomic) centers. While this has the disadvantage that the granularity is fixed, the granularity is sufficient for our initial target machines (Intel Touchstone Delta, Kendall Square Research, and IBM SP1), and the scheme has the advantage that the sparsity may be used in the outer strip-mining loops. The simplest parallel loop structure is presented in Figure 3.

The four loops over *iat*, *jat*, *kat*, and *lat* determine unique quartets of atoms, and the outermost two IF blocks take advantage of sparsity using the Schwarz inequality (the screening information is compressed and may be distributed). Parallelism with full dynamic load balancing is introduced by comparing a sequential count of interacting atomic quartets (*ijkl*) against the value of a shared counter accessed by calling the function `next_task()`. Each task requires fetching up to six blocks of the density matrix and accumulating into the corresponding blocks of the Fock matrix.

Following our previous analysis,¹⁵ communication may be reduced by up to a factor of 2 by defining a task as all *lat* for given *iat*, *jat*, and *kat*. With this task definition, the *ij*, *ik*, and *jk* blocks of the matrices are common to all atomic quartets in a task. Caching of the blocks is preferred to prefetching for its simplicity and so that communication takes full advantage of sparsity. The larger tasks also eliminate a potential bottleneck accessing the shared counter. However, the available parallelism is limited by this choice, and the largest task size increases linearly with the number of atoms in the molecule, which causes

```
task = next_task()
ijkl = 0
DO iat = 1, Natom
  DO jat = 1, iat
    IF (T(iat,jat) .gt. tol1) then
      DO kat = 1, iat
        lat_hi = kat
        IF (kat .eq. iat) lat_hi = jat
        DO lat = 1, lat_hi
          IF (T(iat,jat)*T(kat,lat) .gt. tol2) then
            IF (ijkl .eq. task) then

              Fetch atomic blocks of density matrix
              Dij, Dik, Dil, Djk, Djl, Dkl

              Compute integrals and form Fock matrix

              Update atomic blocks of Fock matrix
              Fij, Fik, Fil, Fjk, Fjl, Fkl

              task = next_task()

            ENDIF
            ijkl = ijkl + 1
          ENDIF
        ENDDO lat
      ENDDO kat
    ENDIF
  ENDDO jat
ENDDO iat
```

FIGURE 3. Pseudocode representing the loop structure of the simplest strip-mined parallel two-electron Fock matrix construction.

load-balancing problems. Thus, we actually define a task to be for given *iat*, *jat*, and *kat* up to five values of *lat*, five being a compromise between a large value for optimal caching and a small value for fine-grain load balance. Finally, contention for access to global array elements is reduced and load balancing is further improved by reversing the loop order and randomizing the order of atoms in the input. The final algorithm is given in Figure 4.

Since the global array tools provide direct support for both data-parallel and task-parallel operations, it was straightforward to parallelize the remainder of the SCF program. Thus, all computational steps of complexity greater than $O(N_{\text{atom}})$ are parallelized (essentially only the input and output phases are sequential). Efficient parallel matrix multiply operations are implemented directly using the global array tools, while other operations (e.g., matrix diagonalization) internally convert to the data layout required by the parallel linear algebra routines. In contrast to the Fock matrix construction, the diagonalizer is paral-

```

task = next_task()
ijkl = 0
DO kat = Natom, 1, -1
  DO iat = Natom, kat, -1
    DO jat = iat, 1, -1
      IF (T(iat,jat) .gt. tol1) then
        lat_hi = kat
        IF (kat .eq. iat) lat_hi = jat
        DO lat_lo = 1, lat_hi, 5
          IF (ijkl .eq. task) then
            DO lat = lat_lo, MIN(lat_lo+4, lat_hi)
              IF (T(iat,jat)*T(kat,lat) .gt. tol2) then

                Fetch only if changed Dij, Dik, Djk
                (also storing corresponding Fock elements)

                Fetch Dil, Djl, Dkl

                Compute integrals and form Fock matrix

                Update Fil, Fjl, Fkl

              ENDDO lat
            ENDDO iat
          task = next_task()
        ENDDO jat
      ENDDO iat
    ENDDO kat
  ENDDO kat

```

FIGURE 4. Pseudocode representing the loop structure of the final strip-mined parallel two-electron Fock matrix construction.

lelized by using conventional message passing instead of the global array routines. This strategy, which is feasible because of the regularity of the linear algebra algorithms, produces somewhat higher efficiency.

We emphasize that the combination of asynchronous global array access and conventional message passing produces a hybrid communication strategy that allows optimizing different parts of the application in different ways. Those portions of the application that are not communication intensive but have many different task sizes and unpredictable communication patterns are best implemented with asynchronous arrays; conventional message passing may be more appropriate to those portions that are communication intensive but regular. We anticipate that such hybrid schemes will become increasingly common.

PERFORMANCE MODEL

Let the basis function on each atom overlap on average with functions on α other atoms. The factor α (between 1 and N_{atom}) is determined by

the molecular geometry, diffuseness of the basis set, and integral screening thresholds. There will be $(\alpha N_{\text{atom}})^2/8$ interacting atomic quartets. If each integral takes β seconds to compute and perfect load balance is achieved, then the total computational cost on P processors is approximately

$$T_{\text{compute}} = \frac{(\alpha N_{\text{atom}})^2}{8} \frac{\beta}{P} \left(\frac{N_{\text{basis}}}{N_{\text{atom}}} \right)^4 \quad (2)$$

On average, about four blocks of the density and Fock matrices must be accessed for each quartet. The communication cost is thus approximately (neglecting possible contention and queuing)

$$T_{\text{communicate}} = \frac{(\alpha N_{\text{atom}})^2}{8} \frac{8}{P} \left(t_0 + t_1 \left(\frac{N_{\text{basis}}}{N_{\text{atom}}} \right)^2 \right) \quad (3)$$

where t_0 and t_1 are the latency in seconds and transmission cost in seconds per floating point number. The ratio of computation to communication is readily computed as

$$\frac{T_{\text{compute}}}{T_{\text{communicate}}} = \frac{\beta \left(\frac{N_{\text{basis}}}{N_{\text{atom}}} \right)^4}{8 \left(t_0 + t_1 \left(\frac{N_{\text{basis}}}{N_{\text{atom}}} \right)^2 \right)} \quad (4)$$

Appropriate values for the Intel Touchstone Delta and from a double-zeta plus polarization basis set are

- $\beta = 0.0001$ seconds/integral
- $N_{\text{basis}}/N_{\text{atom}} = 10$
- $t_0 = 0.0003$ seconds
- $t_1 = 10^{-6}$ seconds/word

Substituting these numbers, we obtain

$$\frac{T_{\text{compute}}}{T_{\text{communicate}}} = \frac{1}{8(0.0003 + 0.0001)} = 312.5 \quad (5)$$

This high ratio of computation to communication predicts very high parallel efficiency independent of sparsity up to $O(N_{\text{atom}}^2)$ processors. With the current granularity, the communication cost is primarily due to latency. The large ratio between computation and communication justifies the assumption that communication contention is not significant. Contention will only become an issue if the number of processors approaches (on mesh architectures such as the Delta) the square of this ratio. Queuing for data access is empirically observed to be not significant for the algorithm of

Figure 4. We have also investigated various strategies for randomization of tasks which rigorously eliminated queuing; however, we obtained slightly degraded performance due to fluctuations in load balance. Excellent load balance is ensured by arranging for large tasks to occur first, limiting the maximum task size, and using full dynamic load balancing.

The performance model demonstrates that no matter how fast the integrals are computed or how high the latency is for remote data access, the block size may be increased to obtain an efficient and scalable execution. The only constraints are that the blocks can be held in local memory and there are a sufficient number of tasks to maintain parallel efficiency.

PERFORMANCE

Several test calculations have been executed to demonstrate and explore the performance of the new program. The molecules (which were near their equilibrium geometries) and basis sets are briefly described in Table I. Full details of basis sets and geometries are available from the authors upon request.

Figure 5 displays the speedups obtained for the two-electron Fock matrix construction for these systems on the Intel Touchstone Delta. The single processor times for the two largest systems were estimated by assuming that the calculations on 64 processors were executing at 99% efficiency, a value which is predicted by the performance model. We observe that the series of similar molecules C_2H_6 , C_4H_{10} , C_8H_{18} demonstrate best speedups of 31, 81, and 367, respectively, which are very close to $N_{\text{atom}}^2/2$. This finite speedup is due to the available parallelism being exhausted. Speedup degrades before this number of processors is used, because as the number of tasks per

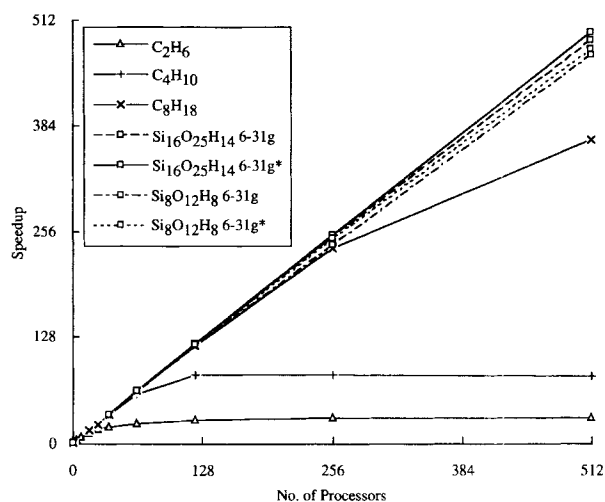


FIGURE 5. Speedup of two-electron Fock matrix construction of the test systems versus number of processors used on the Intel Touchstone Delta.

processor diminishes the load balance worsens. For these small systems, better asymptotic performance could be obtained by varying the maximum task size with the number of processors, so that in the limit of many processors each processor evaluates the interactions of just one quartet of atoms.

Systems with more interactions (e.g., the 28-atom cluster $Si_8O_{12}H_8$, as opposed to the chain C_8H_{18}) or more atoms demonstrate excellent speedup. The largest system, $Si_{16}O_{25}H_{14}$ in a 6-31G* basis, obtains a speedup of 496 on 512 processors, an efficiency of 97%. It is not yet apparent why the efficiency is not higher still, as would be expected from the foregoing simple performance model. One possible cause is a minor bottleneck in the load-balancing mechanism. On 256 processors, the $Si_{16}O_{25}H_{14}$ cluster in a 6-31g basis (461 functions) required just 180 seconds for construction of the Fock matrix with an integral selection threshold of 10^{-7} .

Next to the Fock matrix construction, the other major step is the diagonalization, which, while running in parallel, obtains an effective speedup of only 6. This results in the speedup of the entire SCF calculation being only 438 on 512 processors, an efficiency of 86%. The parallel efficiency of the diagonalization increases as the matrix size increases; however, the $O(N^3)$ scaling of the diagonalization is similar to the scaling of integral evaluation in large molecules. Once improved algorithms are adopted for the Fock matrix construction,²⁻⁵ the diagonalization will be dominant. Several approaches have been proposed for elimi-

TABLE I.
Systems Used to Study Performance of the SCF Program.

Molecule	Basis	Atoms	Functions
C_2H_6	C(5s2p1d)/H(2s1p)	8	64
C_4H_{10}	C(5s2p1d)/H(2s1p)	14	118
C_8H_{18}	C(5s2p1d)/H(2s1p)	26	226
$Si_8O_{12}H_8$	6-31G	28	228
$Si_8O_{12}H_8$	6-31G*	28	348
$Si_{16}O_{25}H_{14}$	6-31G	55	461
$Si_{16}O_{25}H_{14}$	6-31G*	55	707

nating this bottleneck.^{24,25} We are adopting a variation of the second-order convergent approach proposed by Shepard,²⁴ in part because of the wide range of properties that may be computed from the orbital Hessian.

Conclusions

A simple and classical strip-mining algorithm suffices to achieve an efficient and scalable Fock matrix construction in which all matrices are fully distributed. Since the computation is a quartic function of the block size while the communication is only a quadratic function, small block sizes suffice to make the computation time dominate the communication time. A simple performance model that takes into account the cost of integral evaluation, the volume of communication, and the latency and bandwidth of communication predicts that a constant efficiency of about 99% (for the Fock matrix construction) is achieved for the number of processors less than approximately the square of the number of atoms. An efficiency of 97% is measured for a large molecular system on 512 processors. A production version of this algorithm would include dynamic adjustment of the granularity in response to the basis set size and machine performance parameters. Far greater gains are realizable, however, by first pursuing alternative algorithms.²⁻⁵

The current programming model, mixing globally addressable arrays with message passing, is both portable and efficient. In particular, machines such as the Kendall Square Research and the Cray T3D, which have hardware support for shared memory, should prove particularly effective in this model.

Acknowledgments

This work was performed under the auspices of the High Performance Computing and Communications Program of the Office of Scientific Computing, U.S. Department of Energy, under contract DE-AC06-76RLO 1830 with Battelle Memorial Institute, which operates the Pacific Northwest Laboratory, and contract W-31-109-ENG-38 with the University of Chicago, which operates the Argonne National Laboratory. This work has made extensive use of software developed by the Molecular Science Software Group of the Molecular Science Research Center, Pacific Northwest Labora-

tory, which is part of the Environmental and Molecular Science Laboratory. This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Pacific Northwest Laboratory.

References

1. J. Almlöf, K. Faegri, and K. Korsell, *J. Comp. Chem.*, **3**, 385 (1982).
2. I. Panas and J. Almlöf, *Int. J. Quant. Chem.*, **42**, 1073 (1992).
3. R. Dovesi, V. R. Saunders, and C. Roetti, *Crystal 92 User's Manual*, University of Torino, Torino, 1992.
4. M. N. Ringald, M. Belhadj, and R. A. Friesner, *J. Chem. Phys.*, **93** 3397 (1990).
5. O. Vahtras, J. Almlöf, and M. W. Feyereisen, *Chem. Phys. Lett.*, **231** 514 (1993).
6. F. Clementi, G. Corongiu, J. Detrich, S. Chin, and L. Domingo, *Int. J. Quant. Chem. Sym.*, **18**, 601 (1984).
7. M. Dupuis and J. D. Watts, *Theor. Chim. Acta*, **71**, 91 (1987).
8. M. F. Guest, W. Smith, and R. J. Harrison, *Chem. Design Auto. News*, **7**, 12 (1992).
9. A. Burkhardt, U. Wedig, and H. G. v. Schnering, *Theor. Chim. Acta*, **86**, 397 (1993).
10. H. P. Lüthi, J. E. Mertz, M. W. Feyereisen, and J. E. Almlöf, *J. Comp. Chem.*, **13**, 160 (1992).
11. P. Otto and H. Früchtl, *Comp. in Chem.*, **17**, 229 (1993).
12. M. Feyereisen and R. A. Kendall, *Theor. Chim. Acta*, **84**, 289 (1993).
13. M. E. Colvin, C. L. Janssen, R. A. Whiteside, and C. H. Tong, *Theor. Chim. Acta*, **84**, 301 (1993).
14. T. R. Furlani and H. F. King, *J. Comp. Chem.*, **16**, 91 (1995).
15. I. T. Foster, J. L. Tilson, A. F. Wagner, R. Shepard, R. J. Harrison, R. A. Kendall, R. J. Littlefield, D. E. Bernholdt, and J. Anchel, *J. Comp. Chem.*
16. R. H. Harrison and R. A. Kendall, *Theor. Chim. Acta*, **79**, 337 (1991).
17. W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.
18. High Performance Fortran Forum. Technical Report Version 1.0, Rice University, 1993.
19. The MPI Forum. In *Proceedings of Supercomputing '93*, IEEE Computer Society Press, Los Alamitos, CA, 1993, p. 878.
20. J. Nieplocha, R. J. Harrison, and R. J. Littlefield, In *Proceedings of Supercomputing 1994*. IEEE Computer Society, Washington DC, 1994, p. 340.
21. N. Carriero and D. Gelernter, *How To Write Parallel Programs. A First Course*. The MIT Press, Cambridge, MA, 1990.
22. M. Schuler, T. Kovar, H. Lischka, R. Shepard, and R. J. Harrison, *Theor. Chim. Acta*, **84**, 489 (1993).
23. A. P. Rendell, M. F. Guest, and R. A. Kendall, *J. Comp. Chem.*, **14**, 1429 (1993).
24. R. Shepard, *Theor. Chim. Acta*, **84**, 343 (1993).
25. W. T. Pollard and R. A. Friesner, *J. Chem. Phys.*, **99**, 6742 (1993).